
Architecture and Design Considerations for Secure Software

Software Assurance Pocket Guide Series:
Development, Volume V
Version 1.0, October 23, 2009



Software Assurance (SwA) Pocket Guide Resources

This is a resource for 'getting started' in selecting and adopting relevant practices for delivering secure software. As part of the Software Assurance (SwA) Pocket Guide series, this resource is offered for informative use only; it is not intended as directive or presented as being comprehensive since it references and summarizes material in the source documents that provide detailed information. When referencing any part of this document, please provide proper attribution and reference the source documents, when applicable.

This volume of the SwA Pocket Guide series focuses on the architecture and design for secure software. It describes the steps and knowledge required to establish the architecture and high-level design for secure software during the Software Development Life Cycle (SDLC).

At the back of this pocket guide are references, limitation statements, and a listing of topics addressed in the SwA Pocket Guide series. All SwA Pocket Guides and SwA-related documents are freely available for download via the SwA Community Resources and Information Clearinghouse at <https://buildsecurityin.us-cert.gov/swa>.



Acknowledgements

The SwA Forum and Working Groups function as a stakeholder mega-community that welcomes additional participation in advancing software security and refining SwA-related information resources that are offered free for public use. Input to all SwA resources is encouraged. Please contact Software.Assurance@dhs.gov for comments and inquiries.

The SwA Forum is composed of government, industry, and academic members. The SwA Forum focuses on incorporating SwA considerations in acquisition and development processes relative to potential risk exposures that could be introduced by software and the software supply chain.

Participants in the SwA Forum's Processes & Practices Working Group collaborated with the Technology, Tools and Product Evaluation Working Group in developing the material used in this pocket guide as a step in raising awareness on how to incorporate SwA throughout the Software Development Life Cycle (SDLC).

Information contained in this pocket guide is primarily derived from the documents listed in the *Resource* boxes that follow throughout this pocket guide.

Special thanks to the Department of Homeland Security (DHS) National Cyber Security Division's Software Assurance team and Robert Seacord, who provided much of the support to enable the successful completion of this guide and related SwA documents.

Resources

- » “Software Security Assurance: A State-of-the-Art Report”(SOAR), Goertzel, Karen Mercedes, *et al*, Information Assurance Technology Analysis Center (IATAC) of the DTIC at <http://iac.dtic.mil/iatac/download/security.pdf>.
- » IEEE Computer Society, “Guide to the Software Engineering Body of Knowledge (SWEBOK)” at <http://www2.computer.org/portal/web/swbok>.
- » “Lessons Learned from Five Years of Building More Secure Software”, Michael Howard, MSDN, at <http://msdn.microsoft.com/en-us/magazine/cc163310.aspx>.
- » “Microsoft Security Development Lifecycle (SDL) – Process Guidance” at <http://msdn.microsoft.com/en-us/library/84aed186-1d75-4366-8e61-8d258746bopq.aspx>.
- » “The Ten Best Practices for Secure Software Development”, Mano Paul, (ISC)².

Overview

The Guide to the Software Engineering Body of Knowledge (SWEBOK) defines the design phase as both “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” and “the result of [that] process.” The software design phase is the software engineering life cycle activity where software requirements are analyzed in order to produce a description of the software’s internal structure that will serve as the basis for its implementation.

The software design phase consists of the architectural design and detailed design activities. These activities follow software requirements analysis phase and precedes the software implementation the Software Development Life Cycle (SDLC).

Software architectural design, also known as top-level design, describes the software top-level structure and organization and identifies the various components. The architectural design allocates requirements to components identified in the design phase. Architecture describes components at an abstract level, while leaving their implementation details unspecified. Some components may be modeled, prototyped, or elaborated at lower levels of abstraction. Top-level design activities include the design of interfaces among components in the architecture and possibly a database design. Documents produced during the architectural design phase can include:

- » Documentation of models, prototypes, and simulations,
- » Preliminary user’s manual,
- » Preliminary test requirements,
- » Documentation of feasibility, and
- » Documentation of the traceability of requirements to the architecture design.

Figure 1 - Architecture Design With Assurance Activities

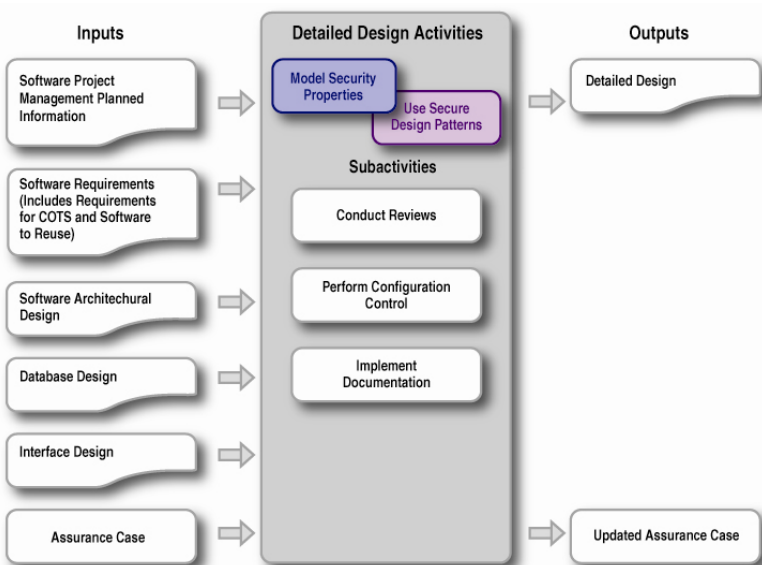
Software detailed design

consists of describing each component sufficiently to allow for its (implementation) construction. Detailed design activities define data structures, algorithms, and control information for each component in a software system. The State-of-the-Art Report SOAR modified Figure 1, illustrates the architectural and detailed design phases of a standard software life cycle depicted in IEEE Standard 1074-2006 with security assurance activities and artifacts included.

To decrease the number of design vulnerabilities, special attention should be devoted to security issues captured during threat modeling, requirement analyses, and early architecture phases. In general, a design vulnerability is a flaw in a software system's architecture, specification, or high-level or low-level design. A design vulnerability is a problem that results from a fundamental mistake or oversight in the design phase. These types of flaws often occur because of incorrect assumptions made about the run-time environment or risk exposure that the system will encounter during deployment.

A MSDN article on "Lessons Learned from Five Years of Building More Secure Software," under the heading "It's not just the code," that highlights many software security vulnerabilities that are not coding issues at all but design issues. When one is exclusively focused on finding security issues in code, that person runs the risk of missing out on entire classes of vulnerabilities. Some security issues in design and semantic flaws (ones that are not syntactic or code related), such as business logic flaws, cannot be detected in code and need to be inspected by performing threat models and abuse case modeling during the design stage of the SDLC.

The best time to influence a project's security design is early in its life cycle. Functional specifications may need to describe security features or privacy features that are directly exposed to users, such as requiring user authentication to access specific data or user consent before use of a high-risk privacy feature. Design specifications should describe how to implement these features and how to implement all functionality as secure features. Secure features are defined as features with functionality that is well engineered with respect to security, such as rigorously validating all data before processing it or cryptographically robust use of cryptographic APIs. It is important to consider security issues carefully and early when you design features and to avoid attempts to add security and privacy near the end of a project's development.



Design Principles for Secure Software

Developers need to know secure software design principles and how they are employed in the design of resilient and trustworthy systems. Two essential concepts of design include abstraction and decomposition of the system using the architecture and constraints to achieve the security requirements obtained during the requirements phase. Most of the readers are probably familiar with these concepts. For those who are not:

Abstraction is a process for reducing the complexity of a system by removing unnecessary details and isolating the most important elements to make the design more manageable.

Decomposition (also known as factoring) is the process of describing the generalizations that compose an abstraction. One method, top-down decomposition, involves breaking down a large system into smaller parts. For object-oriented designs, the progression would be application, module, class, and method.

Other secure software design principles are detailed in a multitude of books, white papers, web portals, and articles. In this section we will provide a brief highlight of two methods and reference resources for additional research. The first one is derived from the developers guide “Enhancing the Development Life Cycle to Produce Secure Software” that describes three general principles summarized on Table 1:

Table 1- Adapted from “ <i>Enhancing the Development Life Cycle to Produce Secure Software</i> ”		
General Principle	Key Practices	Principle Design Conformance
Minimize the number of high-consequence targets	Principle of least privilege	Minimizes the number of actors in the system granted high levels of privilege, and the amount of time any actor holds onto its privileges.
	Separation of privileges, duties, and roles	Ensures that no single entity (human or software) should have all the privileges required to modify, delete, or destroy the system, components and resources.
	Separation of domains	This practice makes separation of roles and privileges easier to implement.
Don't expose vulnerable or high-consequence components	Keep program data, executables, and configuration data separated	Reduces the likelihood that an attacker who gains access to program data will easily locate and gain access to program executables or control/configuration data.
	Segregate trusted entities from untrusted entities	Reduces the exposure of the software's high-consequence functions from its high-risk functions, which can be susceptible to attacks.
	Minimize the number of entry and exit points	This practice reduces the attack surface.
	Assume environment data is not trustworthy	Reduces the exposure of the software to potentially malicious execution environment components or attacker-intercepted and modified environment data.
	Use only safe interfaces to environment resources	This practice reduces the exposure of the data passed between the software and its environment.
Deny attackers the means to compromise	Simplify the design	This practice minimizes the number of attacker-exploitable vulnerabilities and weaknesses in the system.
	Hold <i>all</i> actors accountable	This practice ensures that all attacker actions are observed and recorded, contributing to the ability to recognize and isolate/block the source of attack patterns.
	Avoid timing, synchronization, and sequencing issues	This practice reduces the likelihood of race conditions, order dependencies, synchronization problems, and deadlocks.

Table 1- Adapted from “ <i>Enhancing the Development Life Cycle to Produce Secure Software</i> ”		
General Principle	Key Practices	Principle Design Conformance
Deny attackers the means to compromise (continue)	Make secure states easy to enter and vulnerable states difficult to enter	This practice reduces the likelihood that the software will be allowed to inadvertently enter a vulnerable state.
	Design for controllability	This practice makes it easier to detect attack paths, and disengage the software from its interactions with attackers.
	Design for secure failure	Reduces the likelihood that a failure in the software will leave it vulnerable to attack.

The second secure software design principle is from the highly-regarded paper “*The Protection of Information in Computer Systems*” by Saltzer and Schroeder which identified another set of protection principles that should be applied when designing a secure system, as shown on Table 2.

Table 2- Adapted from Saltzer & Shroeder “ <i>Protection of Information in Computer Systems</i> ”	
Design Principle	What it Means
Economy of mechanism	Keep the design as simple and small as possible.
Fail-safe defaults	Base access decisions on permission rather than exclusion. This principle means that the default is lack of access, and the protection scheme identifies conditions under which access is permitted.
Complete mediation	Every access to every object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance.
Open design	The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords.
Separation of privilege	A protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter with only a single key.
Least privilege	Every program and every user of the system should operate using the least set of privileges necessary to complete the job.
Least common mechanism	Minimize the amount of mechanism common to more than one user and depended on by all users. Every shared mechanism represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.
Psychological acceptability	It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Resources

- » “*Enhancing the Development Life Cycle to Produce Secure Software (EDLC)*”, DHS SwA Forum Process and Practices Working Group, 2008; at <https://buildsecurityin.us-cert.gov/swa/procwg.html>.
- » “Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software”, Software Assurance Workforce Education and Training Working Group, DHS Build Security In (BSI) portal at <https://buildsecurityin.us-cert.gov/daisy/bsi/940-BSI/version/default/part/AttachmentData/data/CurriculumGuideToTheCBK.pdf>.
- » “The Protection of Information in Computer Systems”, Saltzer and Schroeder, at <http://web.mit.edu/Saltzer/www/publications/protection/>.
- » “The Ten Best Practices for Secure Software Development”, Mano Paul, (ISC)².

Architecture and Threat Modeling

Unified Markup Language (UML), developed by the Object Management Group (OMG), is a widely-used specification for modeling software. UML provides the ability to describe the software architecture using various types of diagrams. It provides diagrams to describe application states, information flow, components interaction and more. UML is quite complex and explaining it in depth is beyond the scope of this pocket guide. For additional information, visit the UML resource page or consult one of the multiple books available on the subject. Next is a brief description of some of the diagrams available in UML.

Use case diagrams describe how an application should be used. The emphasis is on what a system does rather than how. Used cases can be represented either text or graphics and there is no restriction on what should include or look like.

A Class diagram gives an overview of a system by showing its classes and the relationships among them. They can be helpful in understanding the how an application is modeled, how classes interact with each other, and relationships in modules for an Object-Oriented design.

A Component diagram describes the relationship of system components (software modules). A component diagram depicts the component interfaces. Examples of components include data bases, web applications, etc.

Other modeling diagrams include:

- » Object diagrams,
- » Sequence diagrams,
- » Collaboration diagrams,
- » Statechart diagrams,
- » Activity diagrams, and
- » Deployment diagrams.

Some people in the SwA community feel that UML does not allow for the capture of security properties and does not include a model for use or abuse cases. Successful techniques for this purpose include threat modeling, data flow diagrams, abuse cases, and attack trees.

Misuse/Abuse Cases – Misuse cases are similar to use cases, except that they are meant to detail common attempted abuses of the system. Like use cases, misuse cases require understanding the services that are present in the system. A use case generally describes behavior that the system owner wants the system to show. Misuse cases apply the concept of a negative scenario—that is, a situation that the system's owner does *not* want to occur. Misuse cases are also known as abuse cases. For an in-depth view of misuse cases, see Gary McGraw's "Misuse and Abuse Cases: Getting Past the Positive" at the BSI portal at <https://buildsecurityin.us-cert.gov/>.

Misuse cases can help organizations begin to see their software in the same light that attackers do. As use-case models have proven quite helpful for the functional specification of requirements, a combination of misuse cases and use cases could improve the efficiency of eliciting all requirements in a system engineering life cycle. Guttorm Sindre and Andreas Opdahl extended use-case diagrams with misuse cases to represent the actions that systems should prevent in tandem with those that they should support for security and privacy requirements. There are several templates for misuse and abuse cases provided by Sindre and Opdahl, and Alexander. Figure 2 is an example of a use/misuse case diagram from Alexander's paper. The high-level case is shown on the left. Alexander indicated that misuse and use cases may be developed in stages, going from system to subsystem levels and lower as necessary. Lower-level cases may highlight aspects not considered at higher levels, possibly forcing re-analysis. The approach is not rigidly top-down but offers rich possibilities for exploring, understanding, and validating the requirements in any direction.

Use cases describe system behavior in terms of functional (end-user) requirements. Misuse cases and use cases may be developed from system to subsystem levels—and lower as necessary. Lower level cases may draw attention to underlying problems not considered at higher levels and may compel system engineers to reanalyze the system design. Misuse cases are not a top-down method, but they provide opportunities to investigate and validate the security requirements necessary to accomplish the system's mission.

As with normal use cases, misuse cases require adjustment over time. Particularly, it is common to start with high-level misuse cases, and refine them as the details of the system are better understood. Determining misuse cases generally constitutes an informed brainstorming activity among a team of security and reliability experts. In practice, a team of experts asks questions of a system's designers to help identify the places where the system is likely to have weaknesses by assuming the role of an attacker and thinking like an attacker. Such brainstorming involves a careful look at all user interfaces (including environmental factors) and considers events that developers assume a person can't or won't do. There are three good starting points for structured brainstorming:

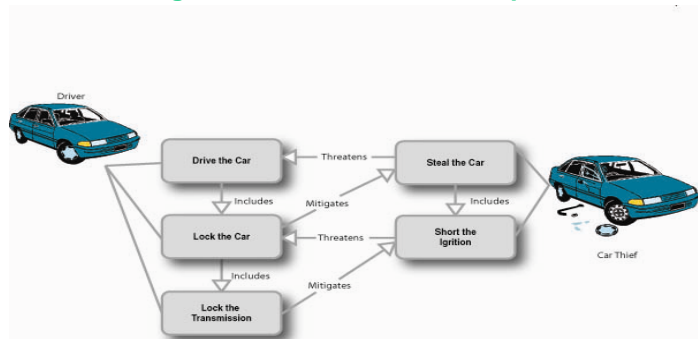
- » First, one can start with a pre-existing knowledge base of common security problems and determine whether an attacker may have cause to think such a vulnerability is possible in the system. Then, one should attempt to describe how the attacker will leverage the problem.
- » Second, one can brainstorm on the basis of a list of system resources. For each resource, attempt to construct misuse cases in connection with each of the basic security services: authentication, confidentiality, access control, integrity, and availability.
- » Third, one can brainstorm on the basis of a set of existing use cases. This is a far less structured way to identify risks in a system, yet it is good for identifying representative risks and for ensuring the first two approaches did not overlook any obvious threats. Misuse cases derived in this fashion are often written in terms of a valid use and then annotated to have malicious steps.

The OWASP CLASP process recommends describing misuse cases as follows:

- » A system will have a number of predefined roles, and a set of attackers that might reasonably target instances of the system under development. Together these should constitute the set of actors that should be considered in misuse cases.
- » As with traditional use cases, establish which actors interact with a use case — and how they do so — by showing a communicates-association. Also as traditionally done, one can divide use cases or actors into packages if they become too unwieldy.
- » Important misuse cases should be represented visually, in typical use case format, with steps in a misuse set off (e.g., a shaded background), particularly when the misuse is effectively an annotation of a legitimate use case.
- » Those misuse cases that are not depicted visually but are still important to communicate to the user should be documented, as should any issues not handled by the use case model.

Threat Modeling - A **threat** is a potential occurrence, malicious or otherwise, that might damage or compromise your system resources. Threat modeling is a systematic process that is used to identify threats and vulnerabilities in software and has become a popular technique to help system designers think about the security threats that their system might face. Therefore, threat modeling can be seen as risk assessment for software development. It enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their limited resources and attention on the parts of the system most “at risk.” It is recommended that all software systems have a threat model developed and documented. Threat models should be created as early as possible in the SDLC, and should be revisited as the system

Figure 2 - Misuse Case example



evolves and development progresses. To develop a threat model, implement a simple approach that follows the NIST 800-30 [11] standard for risk assessment. This approach involves:

- » **Decomposing the application** - understand, through a process of manual inspection, how the application works, its assets, functionality, and connectivity.
- » **Defining and classifying the assets** - classify the assets into tangible and intangible assets and rank them according to business importance.
- » **Exploring potential vulnerabilities** - whether technical, operational, or managerial.
- » **Exploring potential threats** - develop a realistic view of potential attack vectors from an attacker's perspective, by using threat scenarios or attack trees.
- » **Creating mitigation strategies** - develop mitigating controls for each of the threats deemed to be realistic. The output from a threat model itself can vary but is typically a collection of lists and diagrams. The OWASP Code Review Guide at http://www.owasp.org/index.php/Application_Threat_Modeling outlines a methodology that can be used as a reference for the testing for potential security flaws.

On-line Resources

- » OMG Unified Modeling Language (UML) at <http://www.uml.org/>.

Secure Design Patterns

A software design pattern is a general repeatable solution to a recurring software engineering problem. Secure design patterns are descriptions or templates describing a general solution to a security problem that can be applied in many different situations. They provide general design guidance to eliminate the introduction of vulnerabilities into code or mitigate the consequences of vulnerabilities. Secure design patterns are not restricted to object-oriented design approaches but may also be applied to procedural languages. Secure design patterns provide a higher level of abstraction than secure coding guidelines. Secure design patterns differ from security patterns in that they do not describe specific security mechanisms such as access control, authentication, authorization and logging. They define secure development processes, or provide guidance on the configuration of existing secure systems.

The **Secure Design Patterns** technical report categorizes three general classes of secure patterns according to their level of abstraction: architecture, design, and implementation. This section provides a brief summary of the architecture and design patterns. For sample implementation code of these patterns please visit www.cert.org/archive/pdf/09tr010.pdf.

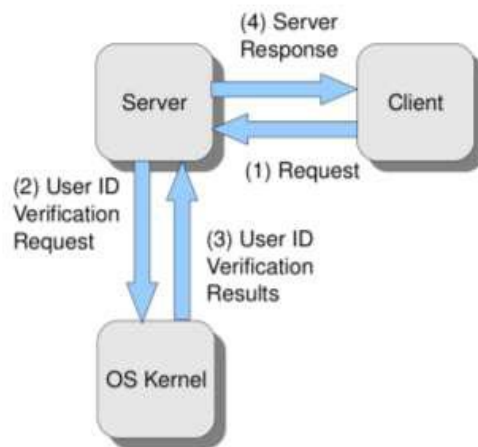
Architectural-level Patterns - Architectural-level patterns focus on the high-level allocation of responsibilities between different components of the system and define the interaction between those high-level components and include:

- » **Distrustful Decomposition** - The intent of the distrustful decomposition secure design pattern is to move separate functions into mutually untrusting programs, thereby reducing the attack surface of the individual programs that make up the system and functionality and data exposed to an attacker if one of the mutually untrusting programs is compromised. This pattern applies to systems where files or user-supplied data must be handled in a number of different ways by programs running with varying privileges and responsibilities.
- » **Privilege Separation (PrivSep)** - The intent of the PrivSep pattern is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the program. The PrivSep pattern is a more specific instance of the distrustful decomposition pattern. In general, this pattern is applicable if the system performs a set of functions that:

- » Do **not** require elevated privileges;
 - » Have relatively large attack surfaces in that the functions;
 - » Have significant communication with untrusted sources; and
 - » Make use of complex, potentially error-prone algorithms.
- » **Defer to Kernel** - The intent of this pattern is to clearly separate functionality that requires elevated privileges from functionality that does not require elevated privileges and take advantage of existing user verification functionality available at the kernel level. The defer to kernel pattern is applicable to systems that:
- » System is run by users who do not have elevated privileges;
 - » Some (possibly all) of the functionality of the system requires elevated privileges; or
 - » Prior to executing functionality that requires elevated privileges, the system must verify that the current user is allowed to execute the functionality.

Figure 3 depicts the general structure of the defer to kernel pattern.

Figure 3 - General Structure of Defer to Kernel Pattern (source: Secure Design Patterns)

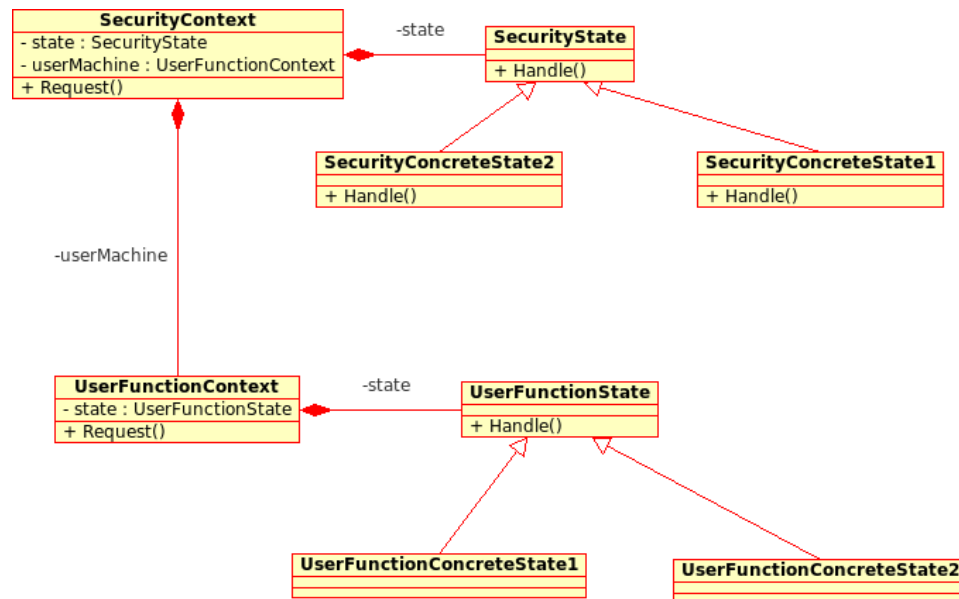


Design-level Patterns - Design-level patterns describe how to design and implement pieces of a high-level system component that addresses problems in the internal design of a single high-level component, not the definition and interaction of high-level components themselves. The design-level patterns defined in this pocket guide include:

- » **Secure State Machine** - The intent of the secure state machine pattern is to allow a clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality as two separate state machines. This pattern is applicable if:
 - » The user-level functionality can be cleanly represented as a finite state machine, or
 - » The access control model for the state transition operations in the user-level functionality state machine can also be represented as a state machine.

The structure of the secure state machine pattern is depicted in Figure 4.

Figure 4 Secure State Machine Pattern (source: Secure Design Patterns)



- » **Secure Visitor** - Secure systems may need to perform various operations on hierarchically structured data where each node in the data hierarchy may have different access restrictions; that is, access to data in different nodes may be dependent on the role/credentials of the user accessing the data. The secure visitor pattern allows nodes to *lock* themselves against being read by a visitor unless the visitor supplies the proper credentials to *unlock* the node. The secure visitor is defined so that the only way to access a locked node is with a visitor helping to prevent unauthorized access to nodes in the data structure. This pattern is applicable if, among other things, the system nodes in the hierarchical data have different access privileges.

Resources

- » "Secure Design Patterns", Chad Dougherty, Kirk Sayer, Robert Seacord, David Svoboda, Kazuya Togashi. Software Engineering Institute at www.cert.org/archive/pdf/09tr010.pdf.
- » "Software Security Assurance: A State-of-the-Art Report" (SOAR), Goertzel, Karen Mercedes, *et al.*, Information Assurance Technology Analysis Center (IATAC) of the DTIC at <http://iac.dtic.mil/iatac/download/security.pdf>.

Formal Methods and Architectural Design

Formal methods are the incorporation of mathematically based techniques for the specification, development, and verification of software. Formal methods, in general, improve software security but can be costly and also have limitations of scale, training, and applicability. To compensate for the limitations of scale, formal methods have been applied to selected parts or properties of a software project, in contrast to applying them to the entire system. As for training limitations, it may be difficult to find developers with the needed expertise in formal logic, the range of appropriate formal

methods for an application, or appropriate automated software development tools for implementing formal methods. Formal methods can be useful for verifying a system. Verification shows that each step in the development satisfies the requirements imposed by previous steps.

Formal methods can be used in the design phase to build and refine the software's formal design specification. Since the specification is expressed in mathematical syntax and semantics, it is precise in contrast to nonformal and even semiformal specifications that are open to reinterpretation.

There are many specification languages available, but are limited in scope to the specification of a particular type of software or system, e.g., security protocols, communications protocols, and encryption algorithms. Examples of specification languages and type include:

- » **Model-Oriented:** Z, Vienna Development Method (VDM);
- » **Constructive:** Program/Proof Refinement Logic (PRL);
- » **Algebraic/Property-Oriented:** Larch, Common Algebraic Specification Language (CASL), OBJ;
- » **Process Model:** Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP); and
- » **Broad Spectrum:** Rigorous Approach to Industrial Software Engineering (RAISE) Specification Language (RSL), LOTOS (language for specifying communications protocols).

Formal Methods in Architectural Design - Formal methods can be used in the architecture phase to:

- » Specify architectures, including security aspects of an architectural design,
- » Verify that an architecture satisfies the specification produced during the previous phase, if that specification itself is in a formal language,
- » Establish that an architectural design is internally consistent,
- » Automatically generate prototypes, and
- » Automatically generate a platform-dependent architecture.

Information Assurance (IA) applications frequently must meet mandatory assurance requirements, and the use of formal methods for IA applications is more prevalent than for many other types of applications. Formal methods are used in assuring IA applications can be used to assure correctness for those willing to incur the costs. In IA applications, formal methods have been used to prove correctness of security functionalities for authentication, secure input/output, mandatory access control and security-related trace properties such as secrecy.

A variety of automated tools are available to assist developers adopting formal methods. Theorem provers are used to construct or check proofs. Theorem provers differ in how much the user can direct them in constructing proofs. Model checkers are a recent class of theorem provers that has extended the practicality of formal methods. Another range of automated tools are associated with model-driven architecture (MDA) and model-driven development (MDD) are considered semiformal rather than formal methods.

The SOAR describes how in *Correctness by Construction*, Anthony Hall and Roderick Chapman describe the development of a secure Certificate Authority, an IA application. The formal top-level specification (architecture design) was derived from the functionality defined in the user requirements, constraints identified in the formal security policy model, and results from the prototype user interface. Praxis used a type checker to automatically verify the syntax in the formal top-level specification and reviews to check the top-level specification against the requirements. The formal security policy model and the formal top-level specification are written in Z, a formal specification language, while the detailed design derived from the top-level specification is written in CSP.

In *Modeling and Analysis of Security Protocols*, Peter Ryan *et al*, describe their use of Failure Divergence Refinement (FDR), a model-checking tool available from Formal Systems Ltd., the Caspar compiler, and CSP. They use these tools to model and analyze a protocol for distributing the symmetric shared keys used by trusted servers and for mutual entity authentication.

Other applications of formal methods are mentioned in *Security in the Software Life Cycle*. These include applications by Kestrel and Praxis where they describe technology that includes Software specification, Language, Analysis, and Model-checking (SLAM), which is Microsoft's model checking tool, the Standard Annotation Language (SAL), and Fugue.

Formal Methods and Detailed Design - The formal methods used in detailed design and implementation usually are different from those used in system engineering, software requirements, and software architecture. Formal methods adopted during earlier phases of the SDLC support the specification of systems and system components and the verification of high-level designs. For architecture design, organizations use model checkers such as VDM, and formal specification languages such as Z. Formal methods commonly used in detailed design and implementation are typically older methods, such as Edsger Dijkstra's predicate transformers and Harlan Mill's functional specification approach.

Formal methods for detailed design are most useful for:

- » Verifying the functionality specified formally in the architecture design phase is correctly implemented in the detailed design or implementation phases; and
- » Documenting detailed designs and source code.

For example, under Dijkstra's approach, the project team would document a function by specifying pre- and post-conditions. Preconditions and post-conditions are predicates such that if the precondition correctly characterizes a program's state on entry to the function, the post-condition is established upon exiting. An invariant is another important concept from this early work on formal methods. An invariant is a predicate whose truth is maintained by each execution of a loop or for all uses of a data structure. A possible approach to documentation includes stating invariants for loops and abstract data types. Without explicit and executable identification of preconditions, post-conditions, and invariants for modules, formal methods in detailed design are most appropriate for verifying correctness when the interaction between system components is predefined and well-understood.

On-line Resources

- » "Software Security Assurance: A State-of-the-Art Report" (SOAR), Goertzel, Karen Mercedes, *et al*, Information Assurance Technology Analysis Center (IATAC) of the DTIC at <http://iac.dtic.mil/iatac/download/security.pdf>.

Design Review

Design reviews should be performed by multiple persons with relevant software security expertise and legitimate stakeholder interests. Formal techniques available include scenario-based reviews that were created for architecture reviews. Reviews including security issues are essential at all levels of design. An independent outside review whenever possible is recommended. Design-related portions of the assurance case should be reviewed as well. Since the best results occur when one develops much of the design assurance case along with the design. Using checklists is also useful.

Verification activities are typically conducted during the design phases at a number of types of reviews:

- » Structured inspections, conducted on parts or views of the high-level design throughout the design phase;
- » Independent verification and validation (IV&V) reviews;
- » A preliminary design review conducted at the end of the architecture design phase and before entry into the detailed design phase; and
- » A critical design review conducted at the end of the detailed design phase and before entry into the coding and unit testing phase.

Design Verification—The design should be verified considering the following criteria:

- » The design is correct and consistent with and traceable to requirements;
- » The design implements proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets, error definition, isolation, and recovery;
- » The selected design can be derived from requirements; and
- » The design implements safety, security, and other critical requirements correctly as shown by suitably rigorous methods.

The decision on which reviews to conduct and their definition usually takes place during the design of development schedule. Such definitions typically include entry criteria, exit criteria, the roles of participants, the process to be followed, and data to be collected during each review. The choice of reviews, particularly those performed as part of IV&V, is partly guided by the evaluation requirements at the Common Criteria Evaluation Assurance Level (EAL). The processes for reviewing the architecture and detailed design should also accommodate later reviews of architecture and design modifications.

Resources

- » “Software Security Assurance: A State-of-the-Art Report”(SOAR), Goertzel, Karen Mercedes, *et al*, Information Assurance Technology Analysis Center (IATAC) of the DTIC at <http://iac.dtic.mil/iatac/download/security.pdf>.

Conclusion

This pocket guide compiles architecture and design software techniques for security and offers guidance on when they should be employed during the SDLC.

The Software Assurance Pocket Guide Series is developed in collaboration with the SwA Forum and Working Groups and provides summary material in a more consumable format. The series provides informative material for SwA initiatives that seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development, acquisition and deployment of trustworthy software products. Together, these activities will enable more secure and reliable software that supports mission requirements across enterprises and the critical infrastructure.

For additional information or contribution to future material and/or enhancements of this pocket guide, please consider joining any of the SwA Working Groups and/or send comments to Software.Assurance@dhs.gov. SwA Forums are open to all participants and free of charge. Please visit <https://buildsecurityin.us-cert.gov> for further information.

No Warranty

This material is furnished on an “as-is” basis for information only. The authors, contributors, and participants of the SwA Forum and Working Groups, their employers, the U.S. Government, other participating organizations, all other entities associated with this information resource, and entities and products mentioned within this pocket guide make no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose, completeness or merchantability, exclusivity, or results obtained from use of the material. No warranty of any kind is made with respect to freedom from patent, trademark, or copyright infringement. Reference or use of any trademarks is not intended in any way to infringe on the rights of the trademark holder. No warranty is made that use of the information in this pocket guide will result in software that is secure. Examples are for illustrative purposes and are not intended to be used as is or without undergoing analysis.

Reprints

Any Software Assurance Pocket Guide may be reproduced and/or redistributed in its original configuration, within normal distribution channels (including but not limited to on-demand Internet downloads or in various archived/compressed formats).

Anyone making further distribution of these pocket guides via reprints may indicate on the pocket guide that their organization made the reprints of the document, but the pocket guide should not be otherwise altered. These resources have been developed for information purposes and should be available to all with interests in software security.

For more information, including recommendations for modification of SwA pocket guides, please contact Software.Assurance@dhs.gov or visit the Software Assurance Community Resources and Information Clearinghouse: <https://buildsecurityin.us-cert.gov/swa> to download this document either format (4"x8" or 8.5"x11").

Software Assurance (SwA) Pocket Guide Series

SwA is primarily focused on software security and mitigating risks attributable to software; better enabling resilience in operations. SwA Pocket Guides are provided; with some yet to be published. All are offered as informative resources; not comprehensive in coverage. All are intended as resources for 'getting started' with various aspects of software assurance. The planned coverage of topics in the SwA Pocket Guide Series is listed:

SwA in Acquisition & Outsourcing

- I. Software Assurance in Acquisition and Contract Language
- II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

- I. Integrating Security in the Software Development Life Cycle
- II. Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
- III. Risk-based Software Security Testing
- IV. Requirements & Analysis for Secure Software
- V. Architecture & Design Considerations for Secure Software
- VI. Secure Coding & Software Construction
- VII. Security Considerations for Technologies, Methodologies & Languages

SwA Life Cycle Support

- I. SwA in Education, Training & Certification
- II. Secure Software Distribution, Deployment, & Operations
- III. Code Transparency & Software Labels
- IV. Assurance Case Management
- V. Assurance Process Improvement & Benchmarking
- VI. Secure Software Environment & Assurance Ecosystem
- VII. Penetration Testing throughout the Life Cycle

SwA Measurement & Information Needs

- I. Making Software Security Measurable
- II. Practical Measurement Framework for SwA & InfoSec
- III. SwA Business Case & Return on Investment

SwA Pocket Guides and related documents are freely available for download via the DHS NCSD Software Assurance Community Resources and Information Clearinghouse at <https://buildsecurityin.us-cert.gov/swa>.